Tampereen yliopisto

JUSSI TUURINKOSKI

VERSIONHALLINNAN TYÖKALUT KETTERÄSSÄ OHJELMISTOKEHITYKSESSÄ

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Maaliskuu 2020

TIIVISTELMÄ

JUSSI TUURINKOSKI: Versionhallinnan työkalut ketterässä ohjelmistokehityksessä (Version Control Tools in Agile Software Development)
Tampereen yliopisto
Tietotekniikan koulutusohjelma
Maaliskuu 2020

Tässä työssä tutkitaan toisen ja kolmannen sukupolven versionhallintajärjestelmiä ja niiden työkalujen tarjoamaa tukea ketterässä ohjelmistokehityksessä. Tavoitteena on kartoittaa kumpi vertailuun valituista versionhallintajärjestelmistä soveltuu paremmin ketterän ohjelmistokehityksen vaatimuksiin. Kumpaakin sukupolvea edustamaan on valittu yksi versionhallintajärjestelmä. Subversion toisen ja Git kolmannen sukupolven versionhallintajärjestelmistä. Kontekstia pohjustetaan esittelemällä lukijalle ohjelmistotuotannon perusteita syventymällä ketterään ohjelmistokehitykseen. Lisäksi tunnistetaan ohjelmistokehitykseen liittyviä haasteita. Tämän jälkeen käydään läpi versionhallinnan historiaa, käyttökohteita ja tavoitteita. Suurin ero toisen ja kolmannen sukupolven järjestelmillä on niiden verkostoitumisrakenne. Toinen sukupolvi nojaa keskitettyyn verkostoon, kun taas kolmas sukupolvi toteuttaa hajautettua verkostoa.

Kumpikin työssä kuvattu versionhallintajärjestelmä on erittäin suosittu ja edelleen aktiivisessa käytössä. Kumpikin järjestelmä on toimiva työkalu lähdekoodin versionhallintaan, mutta Gitin joustava varastohierarkia soveltuu paremmin ketterän ohjelmistokehityksen vaatimuksiin. Työnkulkua helpottavat ominaisuudet ja hajautettu verkosto lyövät Subversionin keskitetyn verkoston rajalliset mahdollisuudet. Subversionin puolesta puhuu kuitenkin sen helppokäyttöisyys Gitiin verrattuna sekä pidempi historia. Versionhallintajärjestelmän valitsemisessa tuleekin ottaa huomioon muitakin seikkoja kuin vain työkalujen kyvykkyys, kuten esimerkiksi opittavuus ja integraatiotuki muihin järjestelmiin. Päätös tulee siis tehdä tapauskohtaisesti, eikä absoluuttista oikeaa vastausta ole olemassa.

SISÄLLYSLUETTELO

1. JOHDANTO

Ohjelmaa tai ohjelmistoa kehitettäessä ei ole epätavallista, että päivityksen yhteydessä julkaistaan enemmän kuin yksi uusi toiminnallisuus tai parannus jo olemassa olevaan toiminnallisuuteen. Eri toiminnallisuudet voidaan jakaa omiin projekteihinsa, joissa on omat projektiryhmät. Mahdollisimman sujuva rinnakkaisesti tapahtuva kehitystyö tuotteen lähdekoodiin on tärkeää sen jatkokehityksessä.

Lähdekoodi ja dokumentit ovat olennainen osa ohjelmistoprojektin lopullista tuotetta. Versionhallintatyökalut auttavat näiden tiedostojen hallinnoimisessa. Versionhallinta onkin ohjelmistoprojektin tärkeimpiä projektinhallintatyökaluja. Versionhallinnan perusajatus on erittäin yksinkertainen – ylläpitää tiettyä tilaa tiedostoista ja tiedostorakenteesta. Työkaluissa on kuitenkin eroavaisuuksia itse versionhallinnan lähestymistavasta yksittäisiin ominaisuuksiin. Nämä erot voivat mahdollistaa suuria eroja työskentelytavoissa ja työn kulussa. [1]

Tämän kandidaatintyön tarkoituksena on pureutua ohjelmistokehityksen haasteeseen, jossa samaa ohjelmistoa kehittää samanaikaisesti vähintään kaksi projektiryhmää. Asiaa käsitellään versionhallintajärjestelmien näkökulmasta. Työn suppeudesta johtuen tarkasteltavaksi on valittu kaksi versionhallintajärjestelmää – Subversion ja Git. Tavoitteena on selvittää kumpi versionhallintajärjestelmistä sopeutuu valittuun käyttökontekstiin paremmin.

Luvussa 2 käydään läpi sitä, mitkä ovat ne ohjelmistokehityksen haasteet, joihin tarkasteltavaksi valituilla versionhallintajärjestelmällä pyritään vastaamaan. Luvussa 3 kerrotaan versionhallintajärjestelmien historiasta, käyttökohteista ja tavoitteista. Luku 4 keskittyy työssä esiteltävän toisen sukupolven versionhallintajärjestelmän, SVN:n historiaan ominaisuuksiin ja erilaisiin SVN-palveluihin. Luvussa 5 keskitytään vastaavasti kolmannen sukupolven versionhallintajärjestelmän, Gitin taustoihin ja ominaisuuksiin. Luvussa 6 vertaillaan näiden kahden versionhallintajärjestelmän ominaisuuksia ja peilataan niitä tutkimuskysymykseen. Yhteenvedossa, luvussa 7, käydään läpi tutkimuksen tulokset.

2. OHJELMISTOJEN KEHITTÄMINEN

Ohjelmistokehitys on suunnittelutyön, ohjelmoinnin, dokumentoinnin ja testauksen summa, jonka tuloksena on tuote [2]. Tuote voi olla uusi sovellus, laajennus jo olemassa olevaan sovellukseen tai bugikorjaus. Ohjelmistoprojektissa on yleensä tilaaja ja toimittaja. Tilaaja määrittelee ohjelmiston vaatimukset, jotka tarkentuvat yhteistyössä toimittajan kanssa. Toimittaja vastaa vaatimukset toteuttavan sovelluksen toimittamisesta.

Hyvän ohjelman tai ohjelmiston kehittämisessä vaaditaan paljon tietotaitoa. Tämän vuoksi projektiryhmä koostuu usein eri osa-alueiden asiantuntijoista. Esimerkiksi yksinkertainen puhelinsovellus, joka näyttää säätietoja käyttäjälle, tarvitsee käyttöliittymän, sovelluslogiikan, rajapintaintegraation palveluun, joka tarjoaa säätiedot, sekä mahdollisesti oman tietokannan. Hyvän lopputuloksen edellytys on, että projektiryhmällä on tarvittava asiantuntemus kaikilta näiltä osa-alueilta.

Yhteistyö tilaajan ja toimittajan välillä ei välttämättä pääty heti tuotteen toimituksen jälkeen. Sovellusta voidaan ylläpitää, korjata ja jatkokehittää vielä ensimmäisen toimituksen jälkeen sopimuksesta riippuen. Tämä tulee ottaa huomioon ohjelman sunnitteluvaiheessa. Sovelluksen modulaarisuudella voidaan edesauttaa nopeaa muunneltavuutta uusien tai muuttuneiden vaatimusten toteuttamiseksi. Ohjelman modulaarisuudella tarkoitetaan sitä, että se koostuu mahdollisimman itsenäisistä komponenteista [3]. Mitä modulaarisempi sovellus on, sitä helpompi siihen on tehdä muutoksia, luoda uutta toiminnallisuutta tai integroida se osaksi muita kokonaisuuksia.

2.1 Ketterä ohjelmistokehitys

Perinteisessä ohjelmistokehityksessä kehitystyö pyritään vaiheistamaan eri osakokonaisuuksiin. Pääpaino on hyvässä suunnittelussa ja työn organisoinnissa. Yksi tunnetuimmista ja käytetyimmistä lineaarisista ohjelmistokehitysmenetelmistä on vesiputousmalli (*Waterfall*) [4]. Vesiputousmallista on olemassa useita eri versioita, mutta jokaisessa perusajatus ja työvaiheiden periaatteet ovat samoja. Roycen alkuperäinen malli jaetaan seuraaviin vaiheisiin: järjestelmävaatimusten esittäminen, vaatimusten analysointi, suunnittelutyö vaatimusten pohjalta, koodaus ja itse tuotteen toteutustyö, testaus ja laadunvarmistus sekä asennus, huolto ja ylläpito [5].

Toisin kuin Vesiputousmallissa ketterässä ohjelmistokehityksessä ei pyritä tekemään tarkkaa lopullista suunnitelmaa heti työn alkuvaiheessa, vaan suunnitelma tarkentuu ja muuttuu kehityssyklien myötä työn edetessä. Työ on dynaamisempaa, mikä tekee määrittelyvaatimukseen kohdistuviin muutoksiin reagoimisesta helpompaa [6]. Ketterää ohjelmistokehitystä sovelletaan monin eri tavoin, mutta lähtökohtaisesti se pohjautuu

vuonna 2001 julkaistuun ”*The Agile Manifesto”* -julistukseen. Manifesti sisältää kaksitoista pääkohtaa, jotka on edelleen supistettu neljään arvoon. Nämä arvot ovat vapaasti suomennettuna ihmiset ja vuorovaikutus ennen prosesseja ja työkaluja, toimiva ohjelma ennen perusteellista dokumentaatiota, asiakasyhteistyö ennen sopimusneuvotteluja sekä kyky reagoida muutoksiin ennen suunnitelman noudattamista. [7]

Lean-sovelluskehitys on yksi ketterään kehitykseen liittyvistä näkökulmista. Se pohjautuu japanilaisen autovalmistaja Toyotan tuotantojärjestelmän toimintamalliin. Sen pääperiaatteisiin kuuluu muun muassa turhien asioiden poistaminen. Näitä voivat olla esimerkiksi tuotantoprosessit, jotka eivät tuo tarvittavaa lisäarvoa työlle, vaan ovat enemmän este joustavalle työskentelylle. Tämän lisäksi pääpaino on ryhmätyössä ja tuotteen yhtenäistämisessä. Lisäksi lopullisten päätösten teko tulee jättää niin myöhään kuin mahdollista, mutta sen ajan tullessa ne tulee tehdä mahdollisimman nopeasti. [8]

Leanin lisäksi toinen paljon esillä oleva näkökulma on jatkuva integraatio (*continuous integration*). Samaan asiayhteyteen kuuluvat myös termit jatkuva toimitus (*continuous delivery*), jatkuva testaus (*continuous testing*) ja jatkuva kehitys (*continuous development*). Perimmäisenä ajatuksena on mahdollisimman lyhyt toimitusaika tilauksesta toimitettuun tuotteeseen. [9] Tämä saavutetaan prosessien voimakkaalla automatisoinnilla. Koodimuutosten validointi, testaus ja asentaminen palvelinympäristöön tapahtuvat kaikki yhdestä laukaisimesta, esimerkiksi koodimuutoksen tallentamisesta tiettyyn varastoon. Tämä käynnistää edellä mainitun automatisoidun prosessin.

Ketterän kehitykseen liitetään usein myös käsite MVP (*Minimum Viable Product*). Tällä viitataan tuotteeseen, jolla saavutetaan maksimaalinen asiakaspalaute käytettyyn työmäärään suhteutettuna. [10] Toimitettu tuote toteuttaa todennäköisesti vaaditut funktiot, mutta siinä saattaa olla laatuun liittyviä puutteita. MVP:n avulla voidaan kartoittaa sitä, vastaako tuote tilaajan odotuksia. MVP voidaan ottaa käyttöön esimerkiksi pilottiasiakkaalla, jonka tarjoama palaute auttaa MVP:n kehittämisessä laajan tuotannon tuotteeksi. Tämän lähestymistavan avulla minimoidaan turhan kehitystyön määrää.

2.2 Ohjelmistokehityksen haasteet

Yksi ohjelmistoalan suurista haasteista on tuotteen laadun varmistaminen. Kynetix Technology Groupin toimitusjohtaja, Paul Smyth, nostaa blogissaan esille ohjelmistoalan nuoren iän ja vertaa sitä rakentamisen tuhansien vuosien historiaan. Me teemme tänä päivänä ohjelmistoalalla samoja opettavia virheitä, mitä rakentamisessa tehtiin paljon aikaa ennen Kristusta. Rakennusalalle on ehtinyt kehittyä jo mittava teollisuus valmiiden komponenttien valmistukseen. Tietotaito on aivan eri tasolla. Samanlaista yhtä kehittynyttä, testattua ja hienostunutta kirjoa ei ole sovellusalalla valmiiden kirjastojen ja palveluiden muodossa – moni asia pitää suunnitella ja kehittää itse. Ohjelmat ja ohjelmistot ovat myös paljon kompleksisempia rakennuksiin verrattuna. Toki isossa rakennelmassa pitää ottaa paljon asioita huomioon lujuuslaskennasta paloturvallisuuteen ja moniin muihin eri seikkoihin,

mutta pelkästään yksi koodirivi voi käyttäytyä eri tavalla riippuen lähtöarvoista. On mahdotonta testata kaikki koodi kaikilla mahdollisilla variaatioilla ja silti saavuttaa haluttu tulos toivotussa aikataulussa. Inhimilliset virheet realisoituvat näissä tapauksissa suurella todennäköisyydellä. [11]

Smyth nostaa blogissaan esiin haasteen loppukäyttäjän osallisuuteen ja sitouttamiseen ohjelmistokehitysprojektissa [11]. Olen jo lyhyen kehittäjäurani aikana ehtinyt kokea ja tunnistaa saman ongelman. Palautteen ja testauksen määrä sovelluksen loppukäyttäjältä on erittäin arvokasta, mutta sitä saadaan usein liian vähän ja liian myöhään. Tähän liittyy myös tilaajan tietämättömyys siitä, mitä oikeasti halutaan. Ohjelmistoalan abstraktius on yksi osasyy sille miksi ongelman tunnistaminen on joskus todella haasteellista. Voi olla jokin ongelma, joka halutaan ratkaista sovelluksella, mutta ongelma on voitu tunnistaa väärin [12]. Asia ratkaistaan kuvainnollisesti laittamalla vuotavan katon vuotokohdan alle ämpäri, vaikka oikeasti itse katto pitäisi korjata. Vuorovaikutuksen tuloksena määrittelyvaatimukset muuttuvat lähes poikkeuksetta projektin aikana. Tämä luo oman haasteensa kehitystyölle, kun jatkuvasti pitää huomioida, että matto voidaan vetää jossain vaiheessa alta. Kommunikaatiota projektihenkilökunnan sisällä ja sidosryhmien välillä ei voi korostaa liikaa [13].

Ohjelmistokehitykseen liittyy myös niin sanottu kehittäjän dilemma (*The Developer Dilemma*) [12]. Tämä pohjautuu The National Institute of Standards and Technology:n vuonna 2002 tehtyyn tutkimukseen, jonka mukaan 80 prosenttia kehittäjän ajasta menee vanhojen ongelmien korjaamiseen [14]. Samaan aikaan nykyajan kvartaalitalous luo melko anteeksiantamattomat aikataulupaineet tulosyksiköille. Onkin mielenkiintoista seurata ohjelmistoalan kehitystä, jossa uusia trendejä ja näkökulmia syntyy kuin sieniä sateella. Missä vaiheessa ohjelmistoalalla päästään samaan tilanteeseen, missä muut insinöörialat ovat tänä päivänä? Parhaiden työkalujen ja toimintatapojen löytäminen kestänee vielä useamman sukupolven ajan.

3. VERSIONHALLINTA

Versionhallinnalla tarkoitetaan yleisesti ottaen tiedostovarastoa, jonka muutoksia monitoroidaan ja dokumentoidaan. Järjestelmästä riippuen tiedostoja voidaan muokata joko yhtäaikaisesti tai muokkaamisen aikana tiedostot ovat lukittuina muilta käyttäjiltä. Yhteistä on kuitenkin kyky hallita tiedoston versioita ja dokumentoida niihin tehdyt muutokset. Dokumentointi sisältää pääasiassa aikaleiman muutoksista, käyttäjätiedon, jolla tekijä voidaan tunnistaa sekä tiedon siitä mitä tiedostoissa on muutettu [1].

Visuaalisesti muutokset havainnollistetaan usein rinnakkaisnäkymässä, missä toisella puolella näytetään vanha versio tiedostosta (joko edellinen versio tai vanhempi) ja toisella puolella uusin versio. Tällä tavoin käyttäjä pystyy vertailemaan muutoksia helposti vanhan ja uuden version välillä.

3.1 Versionhallinnan historia

Versionhallinnan noin neljäkymmenvuotinen historia jaetaan karkeasti kolmeen sukupolveen. Sukupolvet erotellaan toisistaan versionhallintajärjestelmien verkostoitumisen, menetelmien ja toiminnallisuuksien luonteen mukaan.

Taulukko 1: Versionhallintajärjestelmien jaottelu kolmeen sukupolveen [15]

Sukupolvi	**Verkosto**	**Toiminnot**	**Samanaikaisuus**	**Esimerkkejä**
Ensimmäinen	Ei ole	Yksi tiedosto kerrallaan	Lukitukset	SCCS, RCS
Toinen	Keskitetty	Usea tiedosto kerrallaan	Merge ennen Commitia	CVS, SourceSafe, TFS, SVN
Kolmas	Hajautettu	Changesets	Commit ennen Mergeä	Mercurial, Bazaar, Git

Ensimmäisen sukupolven versionhallintajärjestelmät ratkaisivat lähdekoodin samanaikaisen muokkaamisen puhtaasti lukitsemalla resurssin. Yhtä tiedostoa pystyi muokkaamaan yksi henkilö kerrallaan, eikä muita muutoksia voinut tehdä prosessin aikana [15]. Esimerkkijärjestelmät tältä aikakaudelta ovat SCCS (esitelty vuonna 1972) [16] ja RCS (esitelty vuonna 1982) [17].

Toisen sukupolven ratkaisut olivat edeltäjiään selvästi kehittyneempiä samanaikaisen kehittämisen näkökulmasta. Versionhallinta oli keskitetty yhteen varastoon, jota pystyi muokkaamaan yhtäaikaisesti useampi henkilö. Ennen tallennusta (*commit*) käyttäjän tuli kuitenkin yhdistää (*merge*) nykyinen työversionsa uusimpaan saatavilla olevaan versioon ennen omien muutosten lisäämistä [15]. Tämän ajanjakson suosituin versionhallinta on Subversion, joka julkaistiin vuonna 2004 [18]. Muita toisen sukupolven järjestelmiä ovat muun muassa Concurrent Versions System, SourceSafe sekä Team Foundation Server.

Kolmas sukupolvi käsittää ajanjakson vuosituhannen alusta aina nykypäivään. Suurimmat erot toisen sukupolven järjestelmiin ovat paikallisuus ja rinnakkaiset säilöt kehityksen apuna. Tallennukset tehdään usein ensin paikalliseen tietosäilöön, jonka jälkeen suoritetaan yhdistäminen yhteiseen työversioon. Maininnan arvoisia kolmannen sukupolven versionhallintajärjestelmiä ovat Mercurial, Bazaar ja Git [15].

3.2 Versionhallinnan käyttökohteet ja tavoitteet

Versionhallintaa käytetään yleensä ohjelmistojen lähdekoodin ylläpidossa, mutta sitä voi soveltaa mielensä mukaan myös muihin käyttötarkoituksiin, missä muutosten dokumentointi ja tiedostojen eri versioiden keskinäinen vertailu ja hallinnointi nähdään tarpeelliseksi. Sen käyttö on olennaista sovelluskehityksessä, missä lähdekoodia muokkaa useampi henkilö. Versionhallinnan käyttö ratkaisee usein kaksi ohjelmistoprojektin haastetta samanaikaisesti – lähdekoodin eri versioiden hallinnan ja muutosten dokumentoinnin. [1]

Pohjimmiltaan kyse on projektinhallinnasta. Versionhallinnan käytön tavoitteena on tehdä ohjelmiston kehitysprosessista ja versioiden ylläpitämisestä mahdollisimman sujuvaa ja helppoa. Ohjelmistoa pystytään kehittämään samanaikaisesti vaikuttamasta muiden sovelluskehittäjien työversioihin [1]. Versionhallinta tekee julkaisujen hallinnasta helppoa, sillä tarvittaessa sovellukseen voidaan tehdä takaisinpäivitys eli *rollback*. Takaisinpäivityksessä otetaan käyttöön vanhempi versio sovelluksesta. Nykypäivänä versionahllintajärjestelmät ottavat myös huomioon työnkulkuun liittyviä asioita ja sopeutuvat helposti osaksi sovelluskehitysprosessia. Ne tarjoavat muun muassa mahdollisuuden kopioida lähdekoodin useisiin kehityshaaroihin, jolloin rinnakkainen kehitys samaan ohjelmistoon on helpompaa.

Lisäksi versionhallintaa voi käyttää koodikatselmointeihin. Katselmoinneissa toinen kehittäjä käy läpi uuden koodin muutokset. Tällä pyritään löytämään yksinkertaisia huolimattomuusvirheitä sekä avaamaan keskustelua käytetyistä suunnitteluratkaisuista [19]. Versionhallintajärjestelmä voi myös olla osana erilaisia integraatioita, jotka auttavat testaamisen ja ympäristöjen päivittämisen automatisoinnissa. Tällöin tallennus voi laukaista tapahtumaketjun. Tapahtumaketju voi koostua esimerkiksi yksikkötesteistä, sovelluksen asentamisesta testipalvelimelle tai lähdekoodin teknisen velan määrän laskemisesta edelliseen tallennukseen verrattuna.

4. SUBVERSION – TIEDOSTOSTA KESKITETTYYN VERKOSTOON

Subversion eli SVN on avoimen lähdekoodin, toisen sukupolven versionhallintajärjestelmä. SVN pohjautuu keskitettyyn versionhallintaverkostoon. Tämä tarkoittaa sitä, että on olemassa yksi ensisijainen varasto, johon kehittäjät tuovat omat muutoksensa. Tässä yhteydessä tallentaminen tarkoittaa muutosten viemistä yksittäisen kehittäjän omasta varastokopiosta keskitettyyn varastoon. Ennen tallentamista viimeisimmät muutokset tulee hakea (*checkout*) paikalliseen työympäristöön. Kun tila on synkronoitu keskitetyn varaston kanssa, omat muutokset voidaan tallentaa keskitettyyn varastoon. [20]

Keskitetty verkosto luo uudet mahdollisuudet ohjelmistokehitykselle, joita edellisen sukupolven versionhallintajärjestelmät eivät ole tarjonneet. Sama lähdekoodi voidaan jakaa useiden kehittäjien kesken ja uudet muutokset saadaan muiden tietoisuuteen välittömästi.

4.1 CVS:stä Subversioniin

Subversion-projekti sai alkunsa vuonna 2000 CollabNet:n toimesta. Tavoitteena oli tuolloin avoimen lähdekoodin versionhallintajärjestelmä, joka mukailisi SVN:n edeltäjän, CVS:n toiminnallisuutta tuomalla kuitenkin uusia ominaisuuksia. Ensimmäinen iso askel tapahtui elokuussa 2001, kun SVN:n lähdekoodi siirtyi CVS-varastosta Subversion-varastoon. Subversionin ensimmäinen virallinen versio 1.0 julkaistiin vuonna 2004. [21]

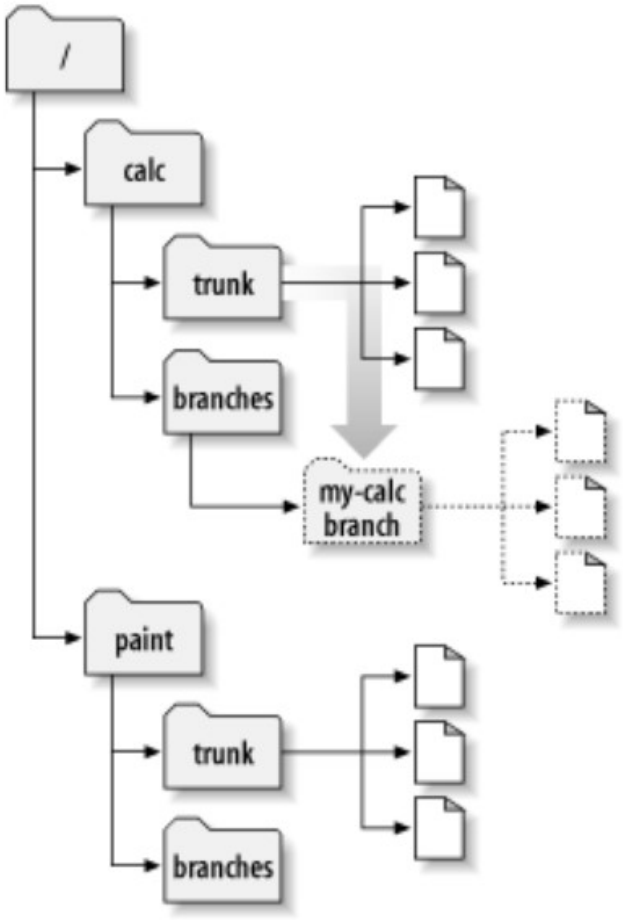
Subversion-projekti toimi alusta alkaen vahvasti tyypillisenä vapaan lähdekoodin projektina, missä palkattuja kehittäjiä oli vain muutama. Vuonna 2009 CollabNetin tavoitteena oli yhdessä Subversion-kehittäjien kanssa saada projekti integroitua osaksi Apache Software Foundationia. Vuoden 2010 alussa Subversion tuli täysin osaksi ASF:n tärkeimpiä projekteja. Sen seurauksena verkkosivusto siirtyi Apachen osoitteen alle ja versionhallintajärjestelmä sai uuden nimen Apache Subversion [21]. SVN:stä on julkaistu uusia versioita tasaisesti viimeisen kymmenen vuoden aikana. Uusin versio on 1.9.4, joka julkaistiin 28. huhtikuuta 2016 [20].

4.2 SVN:n ominaisuuksia

SVN pohjautuu vahvasti edeltäjäänsä CVS:ään. Siinä missä CVS pitää kirjaa yksittäisten tiedostojen muutoshistoriasta, SVN tarjoaa virtuaalisen versioidun tiedostojärjestelmän, joka ylläpitää muutoshistoriatietoja koko kansiorakenteesta. CVS:stä puuttuu kyky ylläpitää tietoja mahdollisista kopioiduista tiedostoista tai tiedostojen nimenmuutoksista. Historiatiedot rajoittuvat tiedoston sisällön muutoksiin. SVN ei ole yhtä rajallinen tämä

suhteen, sillä se ylläpitää historiatietoja myös tiedostojen ja hakemistojen lisäämisistä, kopioinneista, poistamisista ja nimen muutoksista.

SVN siis korjaa paljon CVS:n puutteita. Näiden parannusten lisäksi SVN huomioi myös sovelluskehityksen työnkulkua. Se nimittäin tarjoaa haaroitusominaisuuden avuksi sovelluksen rinnakkaista kehittämistä. Sen sijaan, että uusi kehitystyö tallennettaisiin jatkuvasti samaan kansiorakenteeseen, rinnalle voidaan tehdä kopio, joka mahdollistaa koodin eriyttämisen päähaarasta.



Kuva 1: Kansiorakenne haaroituksen jälkeen. [21]

Haaroituksen myötä rinnakkain tehtävä kehitystyö kohdistetaan sille tarkoitetun kansiorakenteen alle (kuvan branches-kansiot). Päähaara pysyy näin koskemattomana omassa polussaan (kuvan trunk-kansiot). Kun haaraan tehty kehitystyö on valmis, muutokset voidaan yhdistää osaksi päähaaraa. [21]

5. GIT – VERKOSTOHIERARKIAN UUDELLEEN MÄÄRITTELY

Git on avoimen lähdekoodin versionhallintajärjestelmä. Se sopii niin pienten kuin suurtenkin projektien versionhallintaan. Sitä mainostetaan helppokäyttöisenä ja nopeasti opittavana ja sen suorituskyky lokaalisuutensa vuoksi on huippuluokkaa moniin muihin versionhallintajärjestelmiin verrattuna. [22]

Suorituskyvyn ohella Git tarjoaa kirjon ominaisuuksia, jotka mahdollistavat joustavat työskentelytavat. Tässä työssä keskitytään vain merkittävimpiin ominaisuuksiin sovelluskehityksen työnkulun näkökulmasta.

5.1 Gitin historia

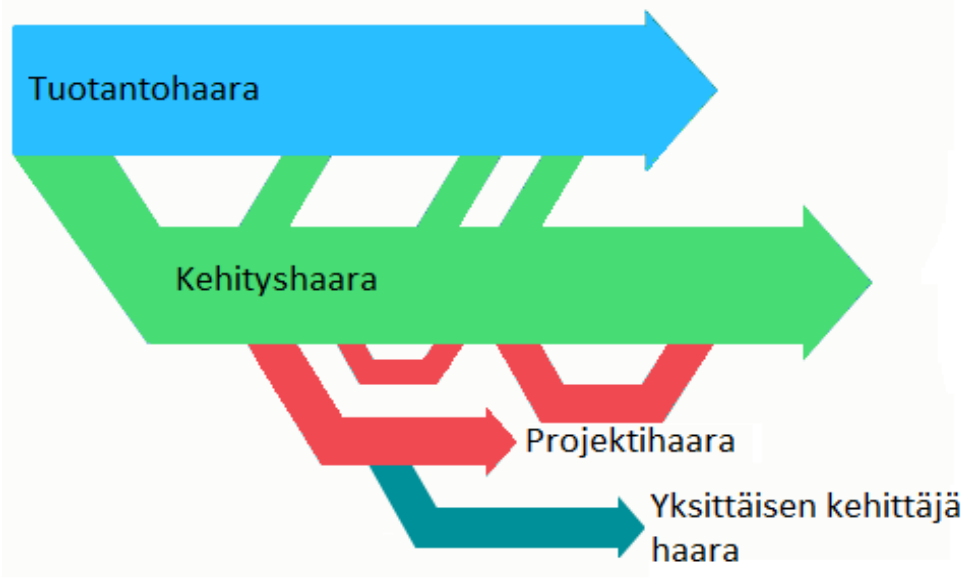
Gitin syntytarina liittyy vahvasti Linux kernelin kehitykseen. Alun alkaen Linux kernelin lähdekoodin versionhallinta oli melko vapaamuotoista. Vuonna 2002 projekti siirtyi käyttämään BitKeeper-nimistä versionhallintajärjestelmää. Käyttö jatkui aina vuoteen 2005 saakka. BitKeeper oli alun perin ilmainen, mutta nyt sen käytöstä alettiin periä maksua. Tämä laukaisi keskustelut uuden versionhallintajärjestelmän käyttämisestä Linux kernel projektissa. Loppujen lopuksi, Linus Torvaldsin johdolla, kehitysryhmä päätyi luomaan oman versionhallintajärjestelmän. BitKeeperin käytön myötä oltiin saatu oppia, jota hyödynnettiin uuden järjestelmän kehityksessä. Päätavoitteet uudessa järjestelmässä olivat nopeus, yksinkertaisuus, tuki ei-lineaariselle kehittämismallille, hajautettu verkosto sekä kyky käsitellä isoja projekteja nopeuden ja tiedon määrän näkökulmasta. [22]

Tämän uuden järjestelmän, Gitin, ensimmäinen versio valmistui huhtikuussa 2005 [release-notes]. Virallista julkaisua saatiin odottaa heinäkuuhun asti. Tämän jälkeen julkaisuja on tapahtunut tasaisesti aina tähän päivään saakka. Viimeisin tuettu versio on 2.4, joka julkaisiin 30. huhtikuuta 2015. Gitin uusin versio, 2.11, on julkaistu 29. marraskuuta 2016.

5.2 Gitin ominaisuuksia

Haaroitus (*branching*) ja yhdistäminen (*merging*) erottaa Gitin monista versionhallintajärjestelmistä. Haaroituksella tarkoitetaan kirjaimellisesti uuden koodihaaran ottamista lähdehaarasta. Uusi haara toimii tämän jälkeen itsenäisesti. Kun kehitys haarassa on valmis, voidaan se yhdistää takaisin lähdehaaraansa. Teknisesti ajateltuna mikään ei estä yhdistämistä myös muihin haaroihin kuin lähdehaaraan, mutta tällöin kannattaa miettiä mitä on tekemässä ja onko se järkevä tapa toimia. Nopea ja helppo kontekstin vaihtaminen mahdollistaa esimerkiksi pienet koodikokeilut. Uuden haaran voi halutessaan hylätä,

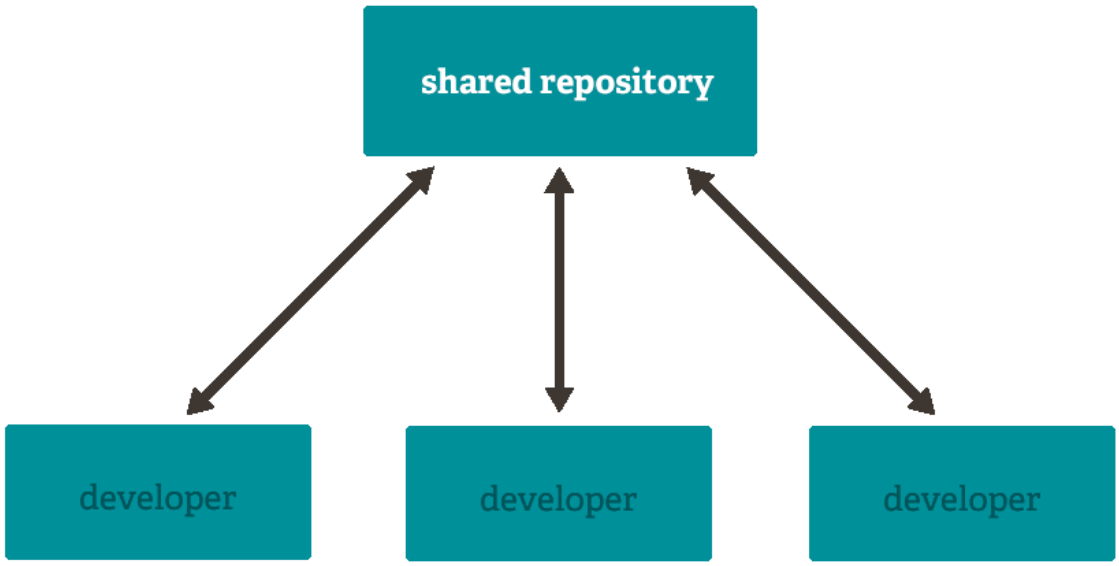
sen muutokset voi yhdistää lähdehaaraan tai sitä voi ylläpitää rinnalla tuomalla uusimmat muutokset lähdehaarasta säännöllisin väliajoin. Haaroitusta voidaan hyödyntää myös koodien roolituksessa. Haara voi edustaa esimerkiksi sovelluksen tuotantoversiota. Tämän rinnalla kulkee kehitysversio omassa haarassa, johon tehdyt muutokset voidaan testata vaikuttamatta tuotantohaaran koodiin. Testatun kehityshaaran muutokset voidaan myöhemmin viedä osaksi tuotantohaaran koodeja esimerkiksi sovelluspäivityksen yhteydessä. Edelleen ajatusta voidaan jatkaa kehityshaarasta omiin toimintohaaroihin. Kehityshaarasta haaroitettu toimintohaara voi toimia yhden projektin pääsääntöisenä kehityshaarana. Tästä edelleen haaroitetut haarat yksittäisen sovelluskehittäjän haaroina.



Kuva 2: Git-haarojen roolitus. [22]

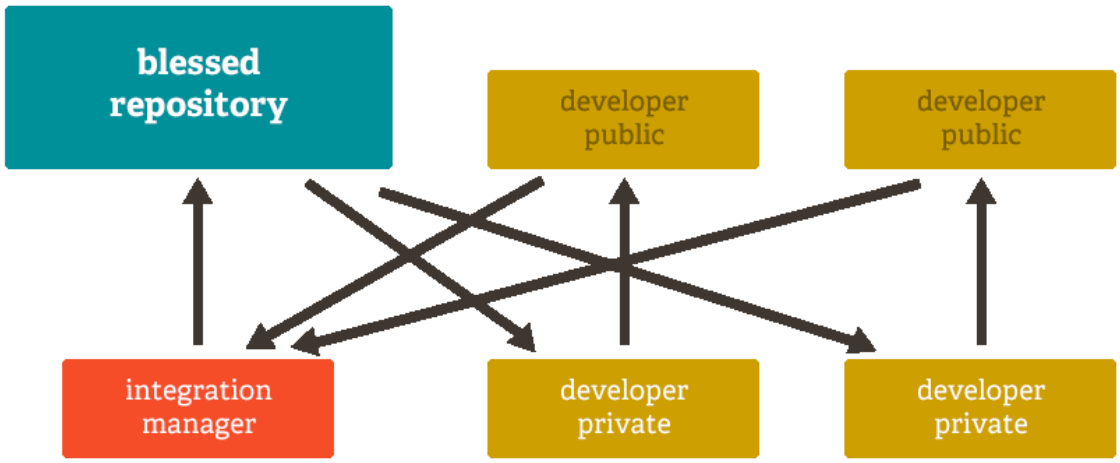
Tarpeen mukaan haaroja voidaan yhdistellä, jotta nähdään muiden kehittäjien muutokset. Haaroitusmenetelmää voidaan toteuttaa niin paljon, kuin se nähdään kyseisessä sovelluskehityksessä tarpeelliseksi. [22]

Kolmannen sukupolven versionhallintajärjestelmän luonteen mukaisesti Git on hajautettu (*distributed*) versionhallintajärjestelmä. Tämä lähestymistapa eroaa toisen sukupolven keskitetyistä versionhallintajärjestelmistä, joissa viimeisimmät muutokset noudetaan keskitetystä varastosta. Hajautetussa järjestelmässä otetaan kokonainen klooni lähdevarastosta. Tämän myötä jokainen varastosta otettu klooni toimii varmuuskopiona lähdevarastolle, sillä teknisestä näkökulmasta katsottuna kloonatut varastot ovat samanarvoisia lähdevaraston kanssa. Hajautettavuuden ja tehokkaan haaroitusominaisuuden johdosta Git taipuu erilaisiin työnkulkutapoihin (*workflow*). Työnkulku voi noudattaa perinteistä keskitetyn versionhallintajärjestelmän mallia, missä kaikki muutokset näkyvät kaikille kehittäjille.



Kuva 3: Keskitetty työnkulku. [22]

Tämä toteutustapa voi olla hyödyllinen, kun ollaan siirtymässä esimerkiksi Subversionin käytöstä Gitiin. Tällöin uuden työkalun opettelu on helpompaa, kun työnkulku on jo tuttu. Toinen erittäin yleinen lähestymistapa on, että yksi henkilö vastaa muutosten tallentamisesta päävarastoon. Tässä mallissa kehittäjä tai projektiryhmä kloonaa alkuperäisen varaston omaan kehitysympäristöön. Toiminnon ollessa valmis, muutokset tallenntaan (*push*) omaan julkiseen varastoon. Tämän jälkeen integraatiosta vastaava taho, esierkiksi laadunvarmistusosasto tallentaa muutokset osaksi päähaaraa.



Kuva 4: Laadunvarmistuksen kautta kulkeva työnkulku. [22]

Tätä lähestymistapaa noudatetaan usein hajautetuissa avoimen lähdekoodin projekteissa. Kehittäjä tai projektiryhmä voi lähettää yhdistämisestä vastaavalle taholle hakupyynnön (*pull request*). Hakupyyntö sisältää usein kehittäjän kommentit sekä kaiken muutoshistorian, mitä ollaan tuomassa osaksi päävarastoa. Hakupyyntö näyttää erot tallennettavan lähdekoodin ja alkuperäisen lähdekoodin välillä. Tämä auttaa koodin katselmoimisessa. Hyväksytyn hakupyynnön jälkeen uusi koodi voidaan yhdistää osaksi alkuperäistä lähdekoodia. [22]

6. VERSIONHALLINTAJÄRJESTELMIEN VERTAILU

Kummankin versionhallintajärjestelmän ominaisuuksia tutkimalla huomaa, että kyseessä on eri aikakausien järjestelmät. SVN:ssä on keskitytty CVS:n luoman pohjan parantamiseen ja laajentamiseen. Kehitystyö on tapahtunut pääasiassa 2000-luvun alkupuolella, jolloin Internet oli hyvin erilainen, mitä se on tänä päivänä viitaten lähinnä verkkoyhteisöjen kasvuun ja sitä kautta syntyneeseen kulttuuriin. Vuosituhannen alussa versionhallinnan kehittäminen painottui perusasioiden parantamiseen, kuten suorituskykyyn, paremman muutoshistoriatiedon ylläpitämiseen ja pelkän tiedoston sijaan koko hakemistorakenteen tilan ylläpitämiseen. Hyvin pian perusominaisuuksien lisäksi tarjottiin tukea myös työnkulun kannalta.

Rinnakkaisen kehittämisen avuksi tuotu haaroitus helpotti työnkulkua niissä tilanteissa, kun samaan sovellukseen tehtiin kehitystä rinnakkain. Oli kyse sitten uusista ominaisuuksista, vanhojen laajentamisesta tai bugien korjaamisesta, lähdekoodiin tehtävät muutokset pystyttiin tekemään eri *”hiekkalaatikoissa”* ja yhdistämään osaksi päähaaran kokonaisuutta myöhemmin, kun se vallitsevan prosessin kannalta oli järkevää. SVN tarjosi toki haaroitusominaisuuden, mutta sen keskitetty verkosto rajoitti haaroituksen käyttömahdollisuuksia. Uusi haara oli vain rinnakkainen kansiorakenne päähaaran rinnalla. Git vei tämän ajatusmallin kuitenkin askeleen pidemmälle tiputtamalla varastohierarkian teknisestä näkokulmasta katsottuna kokonaan pois. Varastoista pystyi tekemään klooneja, jolloin kehitystyö hajauttaminen eri projektiryhmien välillä oli helpompaa. Kehitystyö tapahtui aidosti erillään omissa ympäristöissään. Tämä loi kuitenkin riskin lähdekoodin ristiriitaisuuksille kahden kehitystyön välillä. Pienet ristiriitaisuudet voitiin ratkaista helposti yhdistämisvaiheessa, mutta suuret ristiriitaisuudet vaativat enemmän työtä. Tilanteen varalle on siis hyvä olla olemassa jonkilainen prosessi. Oli se sitten säännöllinen raportointi muutoshistoriasta tai jokin muu sovittu menetelmä.

SVN on järjestelmistä yksinkertaisempi ja helpommin opittava. Se on ollut myös kauemmin olemassa. Monille ohjelmistokehityksen alalla kauemmin toimineille SVN on ollut ensimmäinen versionhallintajärjestelmä. Git on opeteltu vasta sen jälkeen. Ohjelmistoprojektiin työkaluja valittaessa ei voi koskaan korostaa liikaa jo olemassa olevaa teknistä osaamista. Bisnesongelman lisäksi on turha kaivaa itselleen kuoppaa ja luoda tarpeeton tekninen ongelma projektille. Uuden opetteluun menee aikaa, mikä heikentää työn tehokkuutta. Toki asia pätee myös toisin päin. Mikäli Git on järjestelmä, jota on käytetty alusta asti, on se luontevampi valinta ohjelmistokehitysprosessiin opittavuuden näkökulmasta katsottuna.

7. YHTEENVETO

Kuten edellisessä kappaleessa todettiin, versionhallintajärjestelmät edustavat eri sukupolvea ja erilaista verkostohierarkiaa. Versionhallinnan valintaan vaikuttaa muun muassa ohjelmistokehityksessä vallitseva prosessi ja työnkulku. Gitin joustavuus nimenomaan varastohierarkian suunnittelussa tekee siitä suvereenin voittajan tässä suhteessa. Joustavuuden vuoksi Gitin avulla voidaan johtaa selkeämmin työnkulkua ja suunnittelutyötä. Hyvin jäsennetty varastorakenne on suoraa kommunikaatiota projektiryhmän välillä. Tästä voi olla oikein käytettynä suuri apu ohjelmistoprojektissa, sillä suurin osa ohjelmistokehityksen haasteista pystytään ratkaisemaan erinomaisen informaationjaon avulla. Kommunikaatio työryhmän ja sidosryhmien välillä on kaikki kaikessa.

Opittavuuden näkökulmasta SVN:llä on etu. SVN on yksinkertaisempi toiminnaltaan ja se on ollut kauemmin käytössä pidemmän historiansa ansiosta. Sillä on myös parempi integraatiotuki Gitiin verrattuna. Monilla kehittäjillä on aiempaa kokemusta SVN:stä enemmän kuin Gitistä. Tämä on kuitenkin kehittäjäkohtaista. Nykyään tietoa löytyy kuitenkin Internetistä todella helposti ja oppimiskynnys oppaiden avulla on itseasiassa melko pieni. Kummallakin versionhallintajärjestelmällä on takanaan suuri yhteisö, joten tiedon saaminen ei ole ongelma.

Kaiken kaikkiaan tämän työn kontekstiin heijastaen Git soveltuu paremmin ketterän kehityksen vaativiin tarpeisiin. Sen joustavuus nimenomaan hajautettavuuden ansiosta lyö Subversionin tarjoamat edut. Subversion ei kuitenkaan ole huono työkalu, eikä projektin onnistumisen ole suoranaisesti kiinni versionhallintatyökalun valinnasta. Työkalua valittaessa tulisi tunnistaa työnkulkuprosessi, mahdolliset integraatiot muihin järjestelmiin ja jo olemassa oleva tietotaito. On kuitenkin muistettava, että suuremmat haasteet ohjelmistokehityksessä liittyvät kommunikointiin kaikkien sidosryhmien välillä. Tätä ongelmaa ei ratkaista yksin versionhallinnalla.

LÄHTEET

[1] S. Yeates: What Is Version Control? Why Is It Important For Due Diligence? Päivitetty 09.03.2013. Viitattu 23.10.2016. Saatavissa http://oss-watch.ac.uk/resources/versioncontrol

[2] Best Price Computers Articles and Glossary. Viitattu 22.12.2016. Saatavissa http://www.bestpricecomputers.co.uk/glossary/application-development.htm

[3] Metropolia, Ohjelmointi 1 Java luentokalvot. Viitattu 22.12.2016. Saatavissa http://users.metropolia.fi/~miikaval/Ohjelmointi1_Java_virtuaalimateriaali/oppitunti10/modulaarisuus.htm

[4] Selecting a development approach. Päivitetty 27.03.2008. Viitattu 07.12.2016. Saatavissa https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf

[5] W. Royce: Managing the Development of Large Software Systems (1970). Viitattu 07.12.2016. Saatavissa http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[6] C. Larman: Agile and Iterative Development: A Manager's Guide. Addison-Wesley, 2004.

[7] Agile Alliance, Agile 101. Viitattu 07.12.2016. Saatavissa https://www.agilealliance.org/agile101/

[8] M. Poppendieck, Tom Poppendieck: Lean Software Development: An Agile Toolkit. Addison-Wesley, 2003.

[9] Agile Alliance, Lead-Time. Viitattu 22.12.2016. Saatavissa https://www.agilealliance.org/glossary/lead-time/

[10] E. Ries: Minimum Viable Product: a guide. Päivitetty 03.08.2009. Viitattu 07.12.2016. Saatavissa http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html

[11] P. Smyth: 7 Reasons Why Software Development Is So Hard blogikirjoitus. Päivitetty 08.08.2012. Viitattu 23.12.2016. Saatavissa https://www.finextra.com/blogposting/6836/7-reasons-why-software-development-is-so-hard

[12] Coverity Inc., The Software Development Challenge – The Trade-off between Time-to-Market vs. Quality. Päivitetty 2006. Viitattu 23.12.2016. Saatavissa http://www.coverity.com/library/pdf/coverity_bizwhitepaper.pdf

[13] University of Houston, Software Development Challenges. Viitattu 23.12.2016. Saatavissa http://www2.cs.uh.edu/~svenkat/SoftwareDesign/slides/SoftwareDevelopmentChallenges.pdf

[14] National Institute of Standards & Technology, The Economic Impacts of Inadequate Infrastructure for Software Testing. Päivitetty 2002. Viitattu 23.12.2016. Saatavissa https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf

[15] E. Sink: Version Control by Example. 1st Edition, 2011. Viitattu 23.10.2016. Saatavissa http://ericsink.com/vcbe/vcbe_a4_lo.pdf

[16] M. J. Rochkind: The Source Code Control System. IEEE Transaction on Software Engineering, vol. SE-1, No. 4, December 1975. Viitattu 23.10.2016. Saatavissa http://www.basepath.com/aup/talks/SCCS-Slideshow.pdf

[17] GNU. Päivitetty 22.01.2015. Viitattu 25.10.2016. Saatavissa https://www.gnu.org/software/rcs/

[18] Subversion, Release History. Päivitetty 15.09.2015. Viitattu 25.10.2016. Saatavissa https://subversion.apache.org/docs/release-notes/release-history.html

[19] A. Kolawa, D. Huizinga: Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press, 2007.

[20] Apache Subversion. Viitattu 26.10.2016 Saatavissa https://subversion.apache.org

[21] B. Collins-Sussman, B. W. Fitzpatrick, C. M. Pilato: Version Control with Subversion (for Subversion 1.7). O'Reilly 2011.

[22] Git. Viitattu 23.10.2016. Saatavissa https://git-scm.com